Towards JMS Compliant Group Communication – a Semantic Mapping*

Arnas Kupšys Stefan Pleisch André Schiper Matthias Wiesmann
École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
{firstname.lastname}@epfl.ch

Abstract

Group communication provides communication primitives with various semantics and their use greatly simplifies the development of highly available services. However, despite tremendous advances in research and numerous prototypes, group communication stays confined to small niches and academic prototypes. In contrast, message-oriented middleware such as the Java Message Service (JMS) is widely used, and has become a de-facto standard. We believe that the lack of a well-defined and easily understandable standard is the reason that hinders the deployment of group communication systems.

Since JMS is a well-established technology, an interesting solution is to extend JMS adding group communication primitives to it. Foremost, this requires to extend the traditional semantics of group communication in order to take into account various features of JMS, e.g., durable/non-durable subscriptions and persistent/non-persistent messages. The resulting new group communication specification, together with the corresponding API, defines group communication primitives compatible with JMS.

1. Introduction

Group communication (denoted simply by GC hereafter) has been an active area of research for more than a decade. The notion of process groups, with the possibility to multicast messages to the members of a group, was initially proposed in the context of the V System [8], and later extended by the Isis system in the context of failures [7].

GC systems provide *one-to-many* communication primitives with various semantics (e.g., reliable delivery of messages and/or delivery of messages in total order). Such high-level communication abstractions among groups of processes greatly simplifies the development of highly available services (through replication). Yet, despite tremendous advances in research and numerous prototypes, e.g., [5, 20, 23, 10, 6], GC stays confined to small niches and to academic prototypes. One reason for this might be the lack of flexibility of existing GC prototypes. However, recent GC toolkits can be tailored to the application's need [15, 22, 16], but still are not widely adopted in industry. Performance is also sometimes mentioned as a reason for the lack of acceptance of GC. Clearly, increased quality of service with respect to message delivery has a trade-off in performance. However, at the same time middleware systems that suffer from significant performance problems [12] have seen wide industry acceptance.

In contrast to GC, another communication technology has attracted considerable interest: the so-called message oriented middlewares (MOMs) such as MQSeries [17], Tuxedo [4], and Rendezvous [31]. This technology, which provides abstractions for asynchronous message sending, is increasingly used in industry and is now considered to be an integral part of an enterprise computing infrastructure. One of the key reasons for the success of MOMs has been the wide adoption of the Java Message Service interface (JMS) [14] standard. As can be seen also with other technologies (e.g., HTTP and Java in the context of the World Wide Web), standards can be the driving force for the distribution and acceptance of a technology. Unfortunately, no widely accepted standard exists for GC. We believe that the lack of such a standard is one major reason for the limited distribution and acceptance of group communication in enterprise computing infrastructures. It is thus instrumental to define a standard for GC that drives the acceptance of GC in industry.

In this paper, we propose a standard specification and interface for GC. Instead of specifying yet another GC API with probably similarly low chances of becoming a standard as existing GC APIs, we take advantage of the widespread acceptance of JMS and propose to extend the JMS speci-

*Research supported by OFES under contract number 01.0537-1 as part of the IST REMUNE project (number 2001-65002).

Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA'04)
0-7695-2242-4/04 $ 20.00 IEEE

fication with GC. Currently, JMS supports two paradigms: *message queues* and the *publish/subscribe* paradigm. This paper adds group communication as the third paradigm to JMS. The resulting specification and interface is called JMSGroups and should be easily understandable by both the GC community and developers familiar with JMS. As such, it facilitates the acceptance of group communication by a larger community and provides a powerful environment for building fault-tolerant applications. JMSGroups is defined in the spirit of JMS; this forces us to clearly identify the semantic differences between the two technologies and to bridge this semantic gap. As a consequence, we need to address some of the shortcomings of traditional GC over MOMs that have been highlighted in [3], such as process recovery after a crash.

Note that the paper does not describe a GC implementation on top of JMS, or a JMS implementation where GC is used to make the JMS server fault-tolerant, i.e., GC is not visible at all to JMS users; rather, it proposes a specification and an API for a GC system, which is semantically close to JMS. As we will show later in the paper, the implementation of such system can be done using a JMS server. Also note, that we do not discuss in detail how to implement JMSGroups, a topic that is well understood, e.g., [5, 20, 23, 10, 6].

Related Work. Integrating GC with existing middlewares is not a new idea. For example, GC has been integrated to CORBA in the context of object replication, e.g., Object Group Service (OGS) [11], Eternal System [24], Interoperable Replication Logic (IRL) [2] and Electra [19]. In [25] GC is used to implement high-available replicated Enterprise Java Beans (EJB) services, and [18] provides causal ordering for JMS messages. The goal of all these systems differ from the goal of the paper. To our knowledge, the paper is the first to propose an integration of GC into JMS on the specification and API level.

Roadmap. The rest of the paper is structured as follows. Section 2 gives a brief overview of the JMS notions needed to understand the paper. The core contribution of the paper is in Sections 3, 4 and 5. Section 3 discusses the semantic mapping of GC to JMS. In Section 4, we give a formal specification of the mapping. Section 5 presents a JMS compliant API for GC. Section 6 gives a short overview of our first prototype implementation of JMSGroups and Section 7 concludes the paper.

2. Java Message Service

2.1. The architecture

The Java Message Service (JMS) [14] is a part of Sun Microsystem's Java 2 Enterprise Edition [30]; it is a set of interfaces and associated semantics that govern the access to messaging systems. The basic architecture is shown in Figure 1. JMS assumes a central JMS server, which generally acts as the hub for all communications, and has access to stable storage. The server is transparent to the application, composed of the JMS clients (senders of messages and receivers of messages) and a set of application-defined messages. The JMS specification does not define how the server is implemented. It only defines the interfaces and services that the JMS infrastructure must provide.

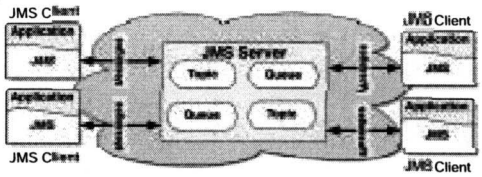

Figure 1. Basic JMS architecture.

Two communication paradigms are defined in the JMS specification: *point-to-point* and *publish-subscribe*. In point-to-point messaging, a message is sent by a JMS client to a specified *message queue*, from which it is extracted by another JMS client (which *consumes* or *receives* the message). Hence, the message sent to a message queue is received by only one client. In contrast, publish-subscribe messaging provides one-to-many communication and is based on the concept of *topic*: a message published by a JMS client to a topic is received by all JMS clients that have subscribed to that topic. Note that the publisher does not know the set of subscribers.

Our proposal is to map the GC semantics and API to the JMS publish-subscribe paradigm. Thus in the next section we focus on this paradigm.

2.2. JMS publish-subscribe

JMS specifies two types of subscriptions to the topic: *non-durable* and *durable*. Consider a topic to which a client has subscribed. With a non-durable subscription the client receives messages published to the topic as long as its connection to the server is active. The connection can break (i.e., become inactive) for example because of a link failure, or because of the crash of the client. Messages published after the connection is broken are not guaranteed to be received by the client.[1] In contrast, durable subscriptions mask these failures. Indeed, the client is ensured to receive all messages that have been published to the topic it

[1] If the connection is broken, the client can try to subscribe again to the topic. Let us assume that the connection was broken at time t_1, and that a new subscription is received by the JMS server at time t_2. With non durable subscriptions, the messages published in the interval $[t_1, t_2]$ may not be received by the client.

has subscribed to, even if the connection is not permanently active.

Another JMS feature is the *message delivery mode*, which can be *persistent* or *non-persistent*. Persistent messages are stored by the JMS server on stable storage, and provide guarantees to publishers in case of the crash of the JMS server. If the JMS server receives a persistent messages, it acknowledges the reception to the publisher only after having stored the message on persistent storage. Non-persistent messages, in contrast, are not saved on persistent storage, and can thus be lost if the JMS server crashes.

Note that durable subscriptions only make sense with persistent messages [14]. In the rest of the paper, we refer to the persistence/non-persistence and the durability/non-durability as quality of service (QoS).

3. GC and JMS: a semantic mapping

3.1. The problem: the semantic gap

JMS and GC have been developed by different research communities, have different semantics and use different APIs. While the problem of mapping one API to the other is not too difficult and will be addressed in Section 5, the semantical mapping raises more difficult and interesting issues. For example, JMS uses notions that are not present in GC, such as durability and persistence. Moreover, JMS assumes a model with process recovery, which is not the case for existing GC systems (where crashed processes recover with a new identity). Consider, for instance, the case of durable subscriptions. Clearly, durable subscriptions only make sense if processes can recover again; a process that does not recover needs no durable subscriptions. As a consequence, in order to support connection durability in GC, we need to change the failure model that underlies GC specifications.

3.2. Bridging the gap

We start the discussion of the semantic mapping between JMS and GC by the key (and simple) idea, which is to represent *process groups* as *JMS topics*.

3.2.1. Mapping groups to JMS topics. We map a group g to a JMS topic t as follows: (i) members of a group g correspond to the subscribers of the corresponding topic t, and (ii) broadcasting a message to the members of g corresponds to publishing the message to the topic t.

The idea of representing a group as a topic is quite natural, since JMS uses the notion of a topic to indirectly address a set of JMS clients. Note that representing the group as a JMS queue is less natural. It raises the following semantic issue: while multiple clients can read from the same queue, only *one* client gets a particular message (i.e., if client c reads message m, then client c' cannot read m). Queues are therefore not suited to express the multicast semantics of GC.

In the context of GC, it is sometimes required that the process that broadcasts a message to group g is a member of g. This is called the *closed* group model. In the *open* group model, no such restriction exists. In JMS a publisher does not have to be a subscriber to publish to the topic. This corresponds to the open group model. Since the open group model is more general than the closed group model, it seems natural to adopt this model for GC based on JMS.

3.2.2. GC message delivery guarantees *vs.* JMS message persistence. In JMS, QoS is defined by *persistence/non-persistence* with respect to messages, and by *durability/non-durability* with respect to subscriptions (see Section 2.2). Although GC does not consider these properties, we show how they can be incorporated in a GC specification. We first address message persistence. Durability is the topic of the next section.

Consider a JMS publisher that publishes message m to topic g. If m is persistent, and the publisher has received an acknowledgment from the JMS server, then the publisher has the guarantee that message m will not be lost, even in case of the JMS server's crash. In contrast, if message m is non-persistent, then m may be lost if the JMS server crashes. Note that the loss of m may occur although the publisher does not crash.

If we transpose the non-persistent case to the context of GC, we have the case of a process that broadcasts some message m to group g with no guarantee that m is ever delivered by any process, even if p does not crash. The message loss does not happen if the message is persistent. In other words, non-persistent messages provide what is usually called *best-effort* guarantees, while persistent messages can be seen as providing the *strong* guarantees of a reliable (logical) channel between the sender and the group. As GC traditionally provides more than best-effort guarantees, we assume persistent messages in the rest of the paper.

3.2.3. GC crash model *vs.* JMS subscription durability. The mapping of durable and non-durable subscriptions is more difficult to address than the question of persistence/non-persistence. The issue cannot be discussed without referring to what happens to the processes that are members of a group and crash.

In one commonly adopted GC model, processes that crash are eventually removed from the group. Upon recovery, these processes adopt a new identity before joining again the group. This model is sometimes called the *crash-no recovery* model: processes that crash seem not to recover, since they recover under a new identity. This model is for example the one of the Isis system [5]. Note that, if

p crashes and later rejoins g with a new identity p', the GC system has no obligation to deliver to p/p' messages broadcast while the process is down.

If we transpose this in terms of type of subscriptions, we see that the crash-no recovery model can very nicely be mapped to non-durable subscriptions, in which the JMS server stops to have any obligation toward a subscriber with respect to message delivery if the connection is broken.

If the crash-no recovery model can be mapped to non-durable subscriptions, what is the GC model that corresponds to durable subscriptions? With durable subscriptions, even if the connection to a subscriber is broken, the JMS server has the obligation to deliver messages to that subscriber. This can be interpreted in the following way in terms of GC. Let p be a process member of group g, and let p crash at time t_1, and later recover at time t_2. Despite of being down during the interval $[t_1, t_2]$, process p delivers all the messages broadcast to the group g. In other words, although p crashes, it is not removed from the group. This means that the GC system has the obligation to deliver to p *all* messages broadcast to g, after p has become a member of g. This model is sometimes called the *crash-recovery* model [1, 27].

3.2.4. GC service *vs.* JMS server. The JMS architecture distinguishes between the JMS server and JMS clients (see Fig. 1). In the context of GC, this distinction is rather unusual. GC specifications usually refers only to what JMS calls *clients*. However, servers cannot be ignored in the context of JMS. Even if this is unusual, it has a positive consequence: it decouples explicitly the *server(s)* that provide the GC service, from the *clients* that use this service. Note that this decoupling does not prevent a process, in some implementation, to be at the same time a client and a server. This special case is the standard case in the context of GC. However, an implementation is not forced to adopt this solution (e.g., [20, 10, 2]). Moreover, an implementation of GC could be based on one single (JMS) server. Of course, such an implementation is not fault-tolerant. Another implementation could be based on multiple (JMS) servers, and so be fault-tolerant. Yet, in another implementation, the same process could be both a (JMS) server and a (JMS) client.

It is important to have the decoupling between clients and servers clear in mind, in order to avoid misunderstanding of some issues discussed below. For example, the distinction between crash-recovery and crash-no recovery can apply both to (JMS) clients, and to (JMS) servers. However, if one model is chosen for (JMS) clients, this does not impose the same model on (JMS) servers. Moreover, in the context of GC specifications, the model issue (crash-recovery $vs.$ crash-no recovery) refers only to (JMS) clients.

4. JMSGroups specification

[13] is usually considered to be a standard specification for GC. However, as discussed in the previous section, JMS introduces features that impact the specification of GC, e.g., the use of stable storage (e.g., message persistence) and the potential recovery of crashed processes (e.g., in the context of durable subscriptions). In this section, we explain how these features impact the specification of GC. We first recall some definitions. We then split the specifications of GC into two parts: (1) the specification of the *reliability guarantees* provided by the broadcast primitive, and (2) the additional *ordering guarantees* that can be superimposed on top of the reliability guarantees provided by the broadcast primitive. Since these two issues are orthogonal, we discuss them separately.

4.1. Definitions

4.1.1. Correct and good processes. In this section, we use the term *process* as a synonym for *JMS client*. A process can be *up* or *down*. A process is up if it is operational, and down if it has crashed. A crashed process, after recovery, is again up. However, the specification of GC is not given in terms of the status up/down of processes at a given time. Instead, the specification refers to the status of processes *over their entire execution*. In this context, many specifications of GC consider that processes do not recover after a crash.[2] In this model, a process that never crashes is said to be *correct* and a process that crashes is said to be *faulty*.

However, because of durable subscriptions, the distinction between correct and faulty processes is not enough. We have to include in our specification the case of processes that crash and later recover. As in [1, 27], we say that a process is *good* if it is eventually always up, i.e., if there is a time t such that after t the process is always up.[3] So, a process that crashes only a finite number of times, and recovers after each crash, is a good process. Trivially, a process that never crashes (i.e., a correct process) is also a good process. Processes that are not good are said to be *bad*.

4.1.2. Membership views. A process group corresponds to a JMS topic. Processes can join a group by subscribing to the corresponding JMS topic; they can leave the group by unsubscribing from the corresponding JMS topic. So, the membership of a group changes over time. In GC, the current group membership is provided to the current group

[2] This does not prevent a process from recovering after a crash. However, the consequence is that a process that crashes must recover under a new identity.

[3] It is usual in a specification to have properties that are eventually true forever. Actually, from a pragmatic point of view, it is sufficient that the property holds "long enough", where "long enough" depends on the application.

members. The information about the current membership of the group is called the group's *view* (of the membership). We do not discuss here the precise specification, we only assume that for every group g, its successive views are totally ordered. Note that this specification is called *primary partition membership* [9]. [4]

4.1.3. Broadcast vs. partial broadcast. A process that broadcasts a message can crash during the execution of the broadcast primitive. This is usually not a problem for specifications. If the broadcast has started, it is considered to be executed; if the sender crashes (during the broadcast or later) there is no obligation for the message to be delivered.

In the context of JMS, the situation is different. This is related to the acknowledgment mechanism provided by JMS (see Sect. 3.2.2). With persistent messages,[5] when some publisher process p (or JMS client) has received an acknowledgment from the JMS server, we have the guarantee that the message will be delivered by the destination processes, *even if p later crashes*. This leads us to distinguish *broadcast* from *partial broadcast*. Consider some process p that broadcasts (i.e., publishes) message m. If p receives the acknowledgment from the JMS server, we say that p has *broadcast* message m. If p crashes before having received the acknowledgment, we say that p has *partially* broadcast message m. Indeed, if no acknowledgment is received by p before the crash, there is no guarantee that the message is received by the JMS server.

4.2. Reliability guarantees of the broadcast primitive

We now formally define the guarantees provided by the broadcast primitive. The properties are expressed in terms of *broadcast* or *partial broadcast*, and *deliver*. [6] Delivery of some message m is the event by which a message is provided to a process (JMS client). We first discuss the case of non-durable subscriptions, and then the case of durable subscriptions.[7] These specifications are adapted from those in [29], which extends the specification in [13] to the case of dynamic groups.

4.2.1. Non-durable subscriptions. In the case of non-durable subscriptions (see Section 3.2.3), the specification distinguishes between correct and faulty processes:

- (P1) *Uniform Validity*: If a process broadcasts message m to the group g, then some *correct* process in g eventually delivers m, or no process in g is *correct*.
- (P2) *Uniform Agreement*: If a process p delivers message m in view v, then all processes that are *correct* in v eventually *deliver* m.[8]
- (P3) *Uniform Integrity*: For any message m, every process in g delivers m at most once, and only if m was previously partially broadcast to g.
- (P4) *Uniform Same View Delivery*: If two processes p and q deliver m, in view v_i for p, and in view v_j for q, then $i = j$.[9]

The Uniform Validity property (P1) is similar to the one in [13]. It is the property we need in the open group model (Sect. 3.2.1), i.e., the model in which the process broadcasting a message to group g does not need to be a member of g. Note that the property is uniform, which means that the delivery is also ensured if the sender crashes after the broadcast has been executed (see the discussion in Sect. 4.1.3).

The Uniform Agreement property (P2) requires agreement on message delivery. While (P1) requires that some correct process delivers the message, (P2) requires that if some process (correct or not) delivers message m, then all correct processes also deliver m.

The Uniform Integrity property (P3) prevents the delivery of duplicate messages. It also requires that the delivery of message m is justified by a corresponding partial broadcast of m. Note that a partial broadcast of m is enough to justify the delivery of m. If a process broadcasts m, and crashes during the broadcast, message m is allowed to be delivered.

The Uniform Same View Delivery property (P4) requires that all processes deliver message m in the same view. This is a standard property in the context of GC. The property is sometimes replaced by a stronger property, called *Sending View Delivery* [9]. However, sending view delivery does not make sense in the open group model.

4.2.2. Durable subscriptions. In Section 3.2.3 we have discussed the link between durable subscriptions and the crash/recovery model. In the case of durable subscriptions, a process p that crashes at time t_1 and recovers at time t_2, after recovery is expected to deliver all messages it has missed in the interval $[t_1, t_2]$. This requirement can only be expressed if the specification distinguishes between good and bad processes, and not only between correct and faulty processes, as for non-durable subscriptions (see Section 4.1.1).

[4] In the specification we assume primary partition membership, but the API presented in the paper can also be applied to partitionable groups.

[5] Recall that we have excluded non-persistent messages from our discussion (Sect. 3.2.2).

[6] We could define *partial deliver* as well, but it does not influence the specification.

[7] To simplify the specifications, we assume here that all members of some group g have the same QoS for the subscription: either all have durable subscriptions, or all have non-durable subscriptions. However, note that in practice different subscription types for the members of the same group are possible.

[8] The notion of *correct in a view* is explained in [29]. It is out of the scope of this paper to discuss this here.

[9] We say that process p delivers message m in view v_i, if the current view of p is v_i when m is delivered.

Hence, for durable subscriptions, we simply replace *correct* by *good* in the properties (P1)-(P4) above (actually only in (P1) and (P2), since (P3) and (P4) do not refer to correct processes).

A comment is needed here for the reader familiar with the GC literature. In most existing GC systems, if process p crashes while in some view v_i, then p is removed from the group. This means that a new view v_{i+1} is defined, from which p is excluded. If p later recovers, and requests to join again, then a new view v_{i+2} is defined, which includes p again. In this case, all messages delivered in view v_{i+1}, will *not* be delivered by p. JMS forces us to handle the case differently with durable subscriptions: *a process p that crashes and later recovers, remains a member of the group, even while being down.* A process is removed from the group only as a result of an *explicit* request to leave the group (i.e., unsubscription from the corresponding topic). This is the behavior that users familiar with JMS expect from a durable subscription, and would be surprised not to have similar guarantees in the context of GC. Our specification enforces this behavior.

4.3. Ordering guarantees of the broadcast primitive

After the specification of the reliability guarantees, we now specify additional ordering guarantees for the delivery of messages. Traditionally, the choice is between no ordering requirement (called *reliable broadcast*), and total order (called *atomic broadcast*).[10]

There is however a more general and elegant solution; the solution consists in using the GC primitive called *generic broadcast* [26]. Generic broadcast orders messages according to a *conflict relation*. Generic broadcast ensures that two messages that conflict are delivered in the same order everywhere. Two messages that do not conflict do not need to be ordered.

For example, we can define that all messages tagged "reliable broadcast" conflict with all messages tagged "atomic broadcast". This ordering guarantee, which is very useful as illustrated in [21, 26], is not provided by the traditional approach. In our specification it can be adapted from [29] as follows:

- (P5) *Uniform Generic Order:* If some process delivers message m in view v before it delivers message m', and the two messages m, m' conflict, then every process p that is in view v delivers m' only after it has delivered m.

Note that specification (P5) is the same for non-durable and durable subscriptions.

[10] We do not discuss causal order here.

For a process p that broadcasts a message to the group g, the "generic broadcast" approach has the following consequence. Instead of choosing a broadcast primitive (reliable broadcast or atomic broadcast), process p simply tags its message with one of the tags defined for group g (there can be more than just two tags). The corresponding conflict relation is attached to the group, and defined at group creation time.

5. Mapping GC primitives to JMS API

The previous section has specified JMSGroups semantics. In this section, we present the JMSGroups API, more specifically, we show the mapping of traditional GC primitives onto the JMS methods related to the publish-subscribe paradigm. Clearly, a direct mapping is not always possible, as some GC concepts do not exist in JMS.

There are two possible approaches here: (1) rely strictly on the interfaces and standard mechanisms offered by JMS, or (2) add new interfaces to JMS where needed (e.g., for functionality specific to GC). Both approaches have advantages and drawbacks. Approach (1) has the important advantage not to modify the existing JMS API, whereas approach (2) violates JMS compatibility and thus might confuse developers familiar with JMS. On the other hand, approach (1) might, for some features, not be very natural from the perspective of GC. Approach (2) does not have this problem.

We have chosen approach (1). By not extending the JMS API for GC, we believe that we increase the acceptance of our proposal. In the following, we show how the additional GC functionality can be mapped onto existing JMS interface methods.

We have to devise a mapping for the GC functionality such as providing views in JMS to group members or issuing the requests to join and remove a process from a group. Moreover, the JMS API does not allow a client p to request a subscription for another client q. In GC systems, a process p can usually issue a request to add another process q to the group. A similar issue arises for leaving the group, in the case of non-durable subscriptions. So we have to provide a mechanism for client p to join another client q, and in the case of non-durable subscription to remove client q.

For this purpose, we use an extension mechanism provided by JMS. Indeed, JMS allows to attach arbitrary *properties* to messages. Using these properties, we can attach membership information to messages and construct special messages to join/remove other members to/from the group. With the same technique, we can map all the GC primitives to the existing JMS API, and remain fully compliant with the JMS API. In the next section we briefly present the JMS classes whose interfaces are relevant in this context.

5.1. Relevant JMS classes and methods

We represent groups as JMS topics and group members as the subscribers to these topics. So, in terms of JMS API, each client is connected to the JMS server using the TopicConnection class. The creation of a TopicConnection instance is expensive (it opens a TCP socket), thus the client usually uses one TopicConnection and from it creates different TopicSession class instances, usually one for each topic it wants to subscribe and/or to publish. Once instantiated the TopicSession class is used to create instances of TopicPublisher and TopicSubscriber classes, which are used to publish and to receive messages, respectively. When creating TopicPublisher and TopicSubscriber the client has to provide a reference to the instance of the Topic class that represents the topic the client is interested in or wants to publish to. Topics are located on the JMS server and their creation is not defined by the JMS specification, i.e., different JMS implementations have their own ways to create topics. Providing the Topic instance reference to the client is also outside the scope of the JMS specification; most implementations for this purpose use Java Naming and Directory Service (JNDI).

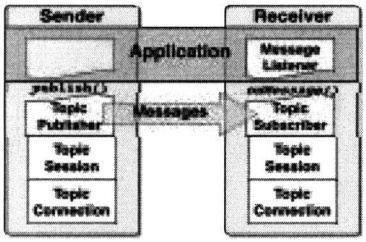

Figure 2. JMS Classes (the arrow shows the logical flow of the messages).

To publish a message to a particular topic the client uses method publish of the TopicPublisher class instance. The reception is done either by (1) calling a method receive on the class TopicSubscriber instance (the call can be blocking if no message is available, or can return immediately), or (2) using a callback. If the client wants to use a callback, it has to provide TopicSubscriber with a reference to a class instance that implements MessageListener interface. Upon message arrival, method onMessage of this interface is called. The messages are created by the TopicSession instance and are represented by the instances of the class Message.

Figure 2 depicts the presented client side classes. We rely on these classes to present the JMSGroups API in the next section.

5.2. JMSGroups API

The JMSGroups methods can be separated into two basic categories: *communication methods* and *administrative methods*. The former are used to set up and maintain the group, while the latter represent the interface used for the actual communication, i.e., for message broadcasts and delivery. Administrative methods and communication methods can further be characterized as *down calls* or *up calls*. Down calls correspond to usual method calls, and up calls correspond to callbacks. Tables 1 and 2 summarize the API mapping, which we now discuss in more details. Where needed, we indicate whether the primitive is a down or up call.

5.2.1. Representing GC's communication primitives using JMS API methods. The GC's communication primitives in the JMSGroups API are represented by the following methods (see Table 1):

Table 1. The Mapping from GC primitives to JMSGgroups API (communication methods).

Communication methods			
GC Primitive	**JMSGroups API**	**Call**	**Description**
broadcast(g, m)	*TopicPublisher.publish(m)*	down	broadcasts a message
deliver(g, m)	*m=TopicSubscriber.receive()*, or *m=TopicSubscriber.receiveNoWait()*	down	delivers a message
	MessageListener.onMessage(m)	up	
viewChange(g,m)	*m=TopicSubscriber.receive()*, with message property *JMS_view* containing new view	down	notifies about a view change
	MessageListener.onMessage(m) with message property *JMS_view* containing new view	up	
getGroupView(m)	*m.getStringProperty("JMS_view")*	down	returns the view in which message m was delivered.
suspect(g, processName)	deliver a special JMS message with a property *JMS_suspect* containing a suspected process name	up	delivers a notification of a suspicion

broadcast(g, m). The *broadcast* primitive sends a message to all members of a group. In order to broadcast a message m to some group g, a client simply calls the method *publish(m)* on the instance of the JMS TopicPublisher class that corresponds to g. The client uses this method to send all messages, regardless of the type of ordering properties he expects (order or no order). The ordering constraints are defined by the message conflict relation (see Sect. 4.3), and

the client just needs to attach the appropriate tag to each message.

deliver(g, m) — down call. In order to deliver a message broadcast to group g, a client simply calls the method *receive()* on the the instance of the JMS TopicSubscriber class that corresponds to g. The call is blocking if no message is available. Note that a non-blocking JMS method, called *receiveNoWait()*, is also available.

deliver(g, m) — up call. To enable the delivery of a message broadcast to group g, a client can also register a callback provided by the JMS interface.

viewChange(g, m). Traditionally GC systems have a special call to notify group members about a view change. However, JMS has no such interface, but allows the attachment of *properties* to messages. So, a simple solution is to consider that delivering a new view v for group g is like delivering a message m for group g. A *view message* is distinguished from a *normal message* by its *JMS_view* property with a value equal to the new view. Similarly to normal messages, a view change message can be received either by a down call or an up call (callback).

getGroupView(m). Traditionally, GC systems provide the means to query the current membership (i.e., view) of the group. JMS does not provide an interface method for this. As for view messages, we propose to attach a property *JMS_view* to the ordinary messages, whose value is the view in which the message was delivered. So, calling the method *m.getStringProperty(JMS_view)* returns the view in which message m was delivered. To get the current view, the client must call this method on the last message delivered, where the last message is either a normal message, or a view message.

suspect(g, processName). Fault detection and suspicion notification are orthogonal issues to GC. The application can have their own fault detection mechanism or use a separate service. But as we mentioned in Section 3.2.4, the JMS server can be seen as an entity providing a GC service for the clients, in some implementation it can be used to provide a fault notification service as well. For such cases we introduce an interface for the suspicion delivery. As we did for the view change, here we also rely on a message with the dedicated properties. If the infrastructure suspects a process in the group g, then a special JMS message must be constructed with a property *JMS_suspect* containing a suspected process name as a value, and broadcast to the group (i.e., published to the topic). The client's reaction to the suspicion is application dependent.

5.2.2. Representing GC's administrative primitives using JMS API methods. The GC's administrative primitives in the JMSGroups API are represented by the following methods (see Table 2):

createGroup(g). Creating a new group corresponds to creating a new JMS topic. As the topic creation is not standardized by JMS specification, every implementation can provide its own mechanism for creating topics (groups).

setMessageConflictRelation(g, conflict). Similarly to the creation of groups, the specification of the message conflict relation for some group g is handled outside of the JMS API. This is done at the group creation time.

joinGroup(g) — non-durable subscription. As explained before, JMS API provides only the interface for a client to subscribe itself to a topic. In the case of non-durable subscriptions, the client calls the JMS method *TopicSession.createSubscriber(g)*, where g is the topic representing a group.

joinGroup(g, processName) — durable subscription.

Table 2. The Mapping from GC primitives to JMSGroups API (administrative methods).

Administrative methods			
GC Primitive	**JMSGroups API**	**Call**	**Description**
createGroup(g)	handled as a creation of a topic and is outside of the scope of the JMS API	–	creates a new group
setMessageConflictRelation(g, conflict)	handled at a group creation time and is outside of the scope of the JMS API	–	defines message conflict relation for group g
joinGroup(g)	*TopicSession.createSubscriber(g)*	down	adds the calling process to the group (non-durable subscription)
joinGroup(g, processName)	*TopicSession.createDurableSubscriber(g, processName)*	down	adds the calling process to the group (durable subscription)
	TopicPublisher.publish(m), where m has the properties *JMS_join_process* (containing process name) and *JMS_subscription* (containing subscription type)	down	adds another process (could be the calling one) to the group
leaveGroup(g)	*TopicSubscriber.close()*	down	removes the calling process from the group (non-durable subscription)
leaveGroup(g, processName)	*TopicSession.unsubscribe(processName)*	down	removes another process (could be the calling one) from the group (durable subscription)
	TopicPublisher.publish(m), where m has the property *JMS_remove*, indicating the name of the process to be removed	down	removes another process (could be the calling one) from the group (non-durable subscription)

Joining a group with a durable subscription requires an additional parameter, namely the process name (i.e., *processName*). In JMS, this parameter is used to uniquely identify a durable subscription. To join a group with a durable subscription, the client thus calls the JMS method *TopicSession.createDurableSubscriber(g, processName)*, where g is the topic representing a group. Note that in this case *processName* corresponds to the name of the process executing this primitive.

joinGroup(g, processName). The subscription of a client to a topic by another client is not specified in JMS. If GC needs this option, we propose the following solution using JMS properties. A special message containing two properties can be used to tell the JMS server that it should invoke a mechanism to subscribe another client. These properties are: *JMS_join_process*, containing a process name, and *JMS_subscription*, containing the subscription type for the newly joined process.

leaveGroup(g) — non-durable subscription. The JMS API provides an interface for the client to unsubscribe itself from the topic. With non-durable subscriptions the client calls the method *close()* on g's instance of the *TopicSubscriber* class in order to leave group g.

leaveGroup(g, processName) — non-durable subscription. In GC, members can generally remove other members from the group, but JMS does not provide a mechanism to remove other subscribers from the topic if they have non-durable subscription. Similarly to *joinGroup(g, processName)* we thus define a special message with a property *JMS_remove* containing the name of the process to be removed. When the JMS server gets such message, it removes the subscription for the specified client.

leaveGroup(g, processName) — durable subscription. For the durable subscriptions, JMS allows a client to unsubscribe another client. To remove a client from the group, the client calls the JMS method *TopicSession.unsubscribe(processName)*. Note that *TopicSession* is not necessary associated with some topic, which implies that *processName* must be unique not only in the group, but in the entire system.

6. Prototype implementation

To validate the concepts presented in this paper, we have implemented a prototype version of JMSGroups. Our implementation relies on a centralised JMS server and extends the JORAM server [28] (see Figure 3 (a)). As the central JMS server is a single point of failure, it is not fault-tolerant. However, it still allows us to validate the proposed semantic mapping and the JMSGroups interface.

The JMSGroups prototype implements most primitives given in Section 5.2, including group view information. Moreover, it handles suspicion notifications and view composition control. Because of the centralised JMS server, all messages are trivially ordered and the ordering comes for free. Hence, the primitive *setMessageConflictRelation* currently has an empty body, i.e., does not take any actions. Also the mechanism for the process to join other processes to the group is not yet implemented; for the moment a process can join only itself.

The removal of a process suspected to have crashed from the group is done as follows. When the JMS server suspects the crash of some process p of group g (i.e., a subscriber), and the subscription is non-durable, then all the members of g are notified. At that point, it is up to the group members to decide whether or not to remove p from g by executing *leaveGroup(g,p)*. If p did not crash and was not removed, then p continues to receive all messages broadcast to g. A state transfer for joining members is implemented using the dedicated JMS messages carrying a state.

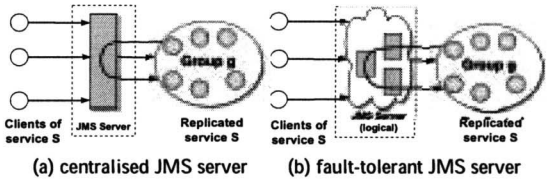

Figure 3. Prototype systems.

We are currently adding fault tolerance to our prototype. This is achieved by replicating the JMS server (see Figure 3 (b)) using protocol stacks already developed in our lab. The stacks provide total order of the messages between JMS servers and support recovery after the crash. The replication of the JMS server is fully hidden, i.e., the members of group g and the clients do not see the difference between (a) and (b) in Figure 3.

7. Conclusion

In this paper we have presented JMSGroups, a novel specification and API for group communication. JMSGroups adds group communication as a third paradigm, besides message queues and publish/subcribe, to standard JMS. For this purpose, we have shown how the features provided by group communication can be mapped onto the standard JMS and thus bridge the semantic gap between GC and JMS. In particular, we have addressed the issue of durable/non-durable subscriptions and persistant messages, concepts that are not available in GC.

As the specification and the interface reflect the spirit of JMS, we hope that our proposal will contribute to a wider use of the group communication abstractions, and that group communication will become an integral part of future applications.

Acknowledgments. We would like to thank Sam Toueg for discussions related to the specification of group communication.

References

[1] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13:99–125, 2000.

[2] R. Baldoni, C.Marchetti, and A.Termini. Active Software Replication through a Three-tier Approach. In *Prooceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02)*, pages 109–118, Osaka, Japan, Oct. 2002. IEEE.

[3] G. Banavar, T. Chandra, R. Strom, and D. Sturman. A Case for Message Oriented Middleware. In *Proceedings of 13th Intl Symposium on Distributed Computing (DISC'99)*, LNCS 1693, pages 1–17, Bratislava,Slovak Republic, 1999. Springer Verlag.

[4] BEA. BEA Tuxedo: The programming model. white paper, BEA Systems, USA, Nov. 1996.

[5] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[6] K. P. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00)*, Hilton Head, South Carolina USA, 2000.

[7] K. P. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of 11th ACM Symposium on Operating Systems Principles*, pages 123–138, 1987.

[8] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems (TOCS)*, 3(2):77–107, May 1985.

[9] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 4(33):1–43, December 2001.

[10] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[11] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

[12] A. S. Gokhale and D. C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks. *IEEE Transactions on Computers*, 47(4):391–413, 1998.

[13] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical Report TR94-1425, CS, University of Toronto; CS, Cornell University, May 1994.

[14] M. Hapner, R. Sharma, J. Fialli, and K. Stout. *JMS specification*. Sun Microsystems Inc., USA, 1.1 edition, April 2002. http://java.sun.com/products/jms/docs.html.

[15] M. Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, Jan. 1998.

[16] M. A. Hiltunen and R. D. Schlichting. The Cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC'00)*, Nürnberg, Germany, 2000.

[17] IBM. *MQSeries Application Programming Guide*. USA, 11 edition, 2000. SC33-0807-10.

[18] P. Laumay, E. Bruneton, N. de Palma, and S. Krakowiak. Preserving causality in a scalable message-oriented middleware. In *Proceedings of the Middleware 2001 (IFIP/ACM)*, volume 2218, pages 311–329, Heidelberg, Germany, November 2001. Lecture Notes in Computer Science, Springer Verlag.

[19] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *USENIX Conference on Object-Oriented Technologies*, 1995.

[20] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phœnix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA, 1995. IEEE. Workshop held during the 7^{th} Symp. on Parallel and Distributed Processing, (SPDP-7).

[21] S. Mena, A. Schiper, and P. Wojciechowski. A Step Towards a New Generation of Group Communication Systems. In *Proceedings of the Int. ACM/IFIP/USENIX Middleware Conference*, LNCS 2672, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.

[22] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the* 21^{st} *International Conference on Distributed Computing Systems (ICDCS'01)*, pages 707–710, Phoenix, Arizona, USA, 2001. IEEE.

[23] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[24] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, USA, September 1999.

[25] M. Pasin, M. Riveill, and T. S. Weber. High-Available Entreprise JavaBeans Using Group Communication System Support. In *Proceedings of the European Research Seminar on Advances in Distributed Systems (ERSADS2001)*, Bologna, Italy, 2001.

[26] F. Pedone and A. Schiper. Handling message semantics with Generic Broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[27] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS'20)*, pages 288–295, Taipe, Taiwan, Apr. 2000.

[28] ScalAgent. *JORAM*. http://joram.objectweb.org.

[29] A. Schiper. Dynamic Group Communication. Technical Report Nbr:200327, École Polytechnique Fédérale de Lausanne (EPFL), 2003.

[30] B. Shannon. *Java 2 Enterprise Edition specification*. Sun Microsystems Inc., USA, 1.4 edition, April 2003.

[31] TIBCO. Rendezvous, TIBCO Messaging Solutions.